AFRL-RI-RS-TR-2016-080

GNOSYS: RAISING THE LEVEL OF DISCOURSE IN PROGRAMMING

NORTHEASTERN UNIVERSITY

MARCH 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-080 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. McKEEVER, JR
Work Unit Manager

/ S /
JOSEPH CAROLI
Acting Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2016	FINAL TECHNICAL REPORT	AUG 2010 – SEP 2015

4. TITLE AND SUBTITLE

GnoSys: RAISING THE LEVEL OF DISCOURSE IN PROGRAMMING

5a. CONTRACT NUMBER
FA8750-10-2-0233

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Olin Shivers

5d. PROJECT NUMBER
CRSH

5e. TASK NUMBER
NO

5f. WORK UNIT NUMBER
RU

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeastern University
360 Huntington Avenue
Boston, MA 02115

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-080

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

GnoSys produced research in the following five areas: 1) A meta-programming system permitting designers to easily construct domain-specific languages for program components; 2) A language suite permitting programs to be written with associated design rationale and behavioral contracts; 3) A high-level operating system factored into distinct modules; 4) A compiler framework and automated reasoning system that could exploit the extra knowledge captured in the form of program annotations, little languages, and component contracts to deliver final systems; and 5) A program-development environment permitting programmers to engage in a "dialogue" with the automated reasoning tools and compiler analyses about the behavior of the computational systems they are designing.

15. SUBJECT TERMS
Formal Methods, meta-programming, domain-specific languages, Software Engineering

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM E. McKEEVER, JR**
U	U	U	SAR	42	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

1. Summary

The GnoSys project's central mantra—something we quoted to each other over and over during the four years of the project to motivate and guide our research—was a single sentence: "Raise the level of discourse in programming."

This mantra sprang from our belief that the central problem with the tools we use today to construct computational systems is their *inexpressivity*. Programmers know a tremendous amount about their programs that is not captured in the program, because their programming language lacks the means to express this knowledge. When multiple programmers develop an application, when an application grows over time, or simply when the complexity of the application overwhelms a single programmer, these tacit assumptions become violated, inviting bugs and creating security holes.

The Clean-slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program's clean-slate mandate gave us an opportunity to synthesize language design and analysis, systems structure, and formal methods in a mutually enabling way into a single, harmonious whole: a language-centered approach to constructing robust systems. The fundamental premise underlying our approach was that **by changing the way programmers specify the systems they are constructing, we can capture more of the important design knowledge—knowledge that can be used to ensure the robustness of the final system.** We set out to build a programming environment to change the way programs are designed, implemented and executed, permitting programmers to capture this missing knowledge in forms that can be exploited for qualitative improvements in robustness and security.

At the outset of the effort, the elements of our system were intended to be:

- A **meta-programming system** permitting designers to easily construct domain-specific languages for program components;
- A **language suite** permitting programs to be written *with associated design rationale* and *behavioral contracts*;
- A **high-level operating system** factored into distinct modules;
- A **compiler framework** and **automated reasoning system** that could exploit the extra knowledge captured in the form of program annotations, little languages, and component contracts to deliver final systems; and

- A **program-development environment** permitting programmers to engage in a "dialogue" with the automated reasoning tools and compiler analyses about the behavior of the computational systems they are designing.

As we'll discuss in this report, we actually did all of this.

In some sense, our intention was to build a Lisp Machine for the new millennium, exploiting the advances in programming-language research that have occurred since the Lisp Machine was designed over 30 years ago. We hoped to construct a system that shares the same kind of advantage in programmer productivity and software assurance over the standard programming systems and execution environments of today that the Lisp Machine enjoyed over the systems of its era. We feel that we did this, as well—and the resulting programming environment is available, for free, on the internet, with open-source licensing.

Where our track record was less than perfect, however, is in the *integration* of all of our research point results into a unified whole. This was really the central point of the GnoSys effort, but also the most difficult to accomplish, structurally. We were able to integrate perhaps 80% of our work—at a university, that's a B grade.

The new technologies that we were able to integrate together were our results in domain-specific language metaprogramming, software contracts, "virtual OS" resource management, and some compiler optimization. The elements that we did *not* manage to integrate are more ambitious results in program analysis and optimization, more foundational metaprogramming work, and gradual typing.

Looking back over the four years of the project, the principal difficulty was simply that some of the more difficult "kernel" results took a full four years to develop simply as point technologies: a DARPA-hard problem worth a doctoral thesis takes about six years to complete, while the CRASH program lasted for four. The process of slotting these results into their proper place in the unified whole is still ongoing as of the date of this final report.

2. Introduction

GnoSys, the system we developed under CRASH, increases the expressiveness of programmers to specify robust, correct systems in three ways.

Domain-specific languages and model-based design First, we developed a new system that permits programmers to capture their designs using custom, domain-specific languages (DSLs). The novelty of our DSL technology is that it permits the language designer to express the rich static semantics associated with DSLs. Thus the benefits of programming in a customized notation are extended not only to *programmers* who read and write programs, but to the *automatic tools* that check these programs for bugs and security violations.

The ability to easily design "little languages" for custom application domains makes it possible for programmers to engage in *model-based design*: by working with a high-level notation, they can use its increased static semantics to make stronger predictions about the programs they write.

A high-level functional operating system Second, we designed and implemented a high-level virtual operating system that expresses basic systems services, such as resource management, task coordination and scheduling, and protection, directly using the linguistic mechanisms of a modern, high-level functional language, such as continuations, modules, contracts, and higher-order functions.

Directly encoding operating-system mechanisms in an advanced language is a strong co-design element of our system: all the advantages of our language technology now apply to the operating system and its client interfaces. The compiler and other program-reasoning tools are now able to interact with the operating system to provide services. Services can easily be virtualized, simply by providing alternative implementations for system interfaces. The operating system can be factored from a vulnerable, monolithic block of code into a set of interacting components, with cross-component interaction managed and checked by our contract technology.

Machine-checkable assertions about programs Finally, we took advantage of the CRASH program's clean-slate opportunity to infuse formal methods pervasively throughout the design. We used declarative, logic-based languages to specify system properties, including contracts, invariants, and models. The freedom to create such logical specifications, in domain-specific terminology, is key to "raising the level of discourse." It is a prerequisite for capturing programmer knowledge about the domain and about the program. Such specification allows the programmer and the compiler to import knowledge about the model. Furthermore, by expressing this knowledge in declarative, logic-based languages, we make it possible for the compiler and other system components to reason about the program and to take advantage of such reasoning both to verify desired properties and to generate potential counterexamples.

Architectural themes

The GnoSys effort concerned itself with the design and implementation of a new software architecture, allowing programmers to write software that can be understood and reasoned about, by both humans and software tools. The design of the architecture followed several themes:

Language-oriented Our approach is fundamentally language-oriented. Our high-level, virtual operating system exploits the idea that "the language is the operating system." Similarly, we see domain-specific languages as a source of reasoning power—a means of capturing more design-time knowledge about a system.

Our goal was to enable programmers to move from *encoding a computation* to *encoding knowledge about a computation.* This shift is what we mean by "raising the level of discourse."

Integrated approach—exploiting co-design We pursued designs that integrate distinct technologies together, in order to optimize global properties of the system. This sort of co-design manifested itself in multiple ways:

- Dynamic and static
  We integrate static reasoning, in the form of program analysis and formal methods, with the expressiveness of dynamic functional languages and contracts.

- Operating system and compiler
  By integrating the compiler with the operating system, traditional OS services can be provided using the linguistic mechanisms of advanced programming languages, such as modules, contracts, automatic resource management, and continuations. This means that client use of OS services can be checked by our tools, increasing program assurance.

- Language and analysis
  We co-design our programming languages and their analyses in tandem. This allows us to use analyses to drive down the cost of expressive language features, or to tune language elements to enable analysis. Even better, we provide this co-design ability to our programmers, as well, using our domain-specific language technology.

- Human programmers and automated reasoning agents
  We develop language and system designs that are intended to facilitate interaction between human programmers and the automated reasoning agents that assist them in delivering high assurance for their programs.

- Cross-disciplinary research team
  Our team was carefully chosen for its cross-disciplinary expertise, which we believe is critical for technology development that is integrated across the technology stack. Members of the team made a point on collaborating on multiple elements of the system, to help ensure that the pieces fit together.

Using what we build The project team has a long track record of doing our research, teaching, and standard programming *within the systems we build*. In particular, we had multi-decade experience living day-to-day "inside" the program-development environment DrRacket and a precursor systems-programming environment Scheme Shell (scsh). We were able to do this pretty consistently throughout the four years of the CRASH program.

Bug-free software is secure and robust Our view is that the way to provide high-assurance software is to provide means of making guarantees about the software. Claiming that bug-free software is secure is true in more ways than one. It is true *in practice*, in that systems are typically penetrated by exploiting bugs in the program, such as buffer overflows or SQL-injection holes. Every time such a bug is eliminated, a security hole is closed. It is also true *by definition*: the

ability to penetrate a system in any way is clearly a bug in either its design or implementation.

Providing mechanisms to ensure that programs satisfy some specification or contract was our consistent means of addressing security and assurance.

Guarding the borders with modules and contracts *Contracts* are a programming-language construct developed by members of the GnoSys team for expressing invariants that govern the flow of all kinds of values (plain values, closures, objects) across an interface between components of a program (procedures, modules, classes). The theory of contracts ensures that they have sensible semantics even in a higher-order setting, where active computation can be passed from one component of a program, through some intermediary, and finally invoked by a third component. Among the subtle issues addressed by contract technology is *blame assignment*, which correctly determines which component is at fault in these complex higher-order cases, when a contract is violated.

It's important to note that contracts are not subject to the static restrictions we place on type systems; they can be any computable property. This is in keeping with the general design philosophy of GnoSys: we put *expressiveness* first, then use static techniques, such as program analysis, to win back efficiency where possible.

Contracts play several important roles in the GnoSys architecture. They are the basic way that programmers in GnoSys express the required invariants and preconditions for a given program component. Once written down, these conditions can be used to protect components from each other. A run-time monitoring system guarantees that if something goes wrong, the violator will be exposed and corrective action taken. In a modular, compartmentalized system such as GnoSys, the run-time system can issue a notification and then convert the violation into a micro-reboot of the associated service.

Contracts also support integration: not only the integration of code components, but the integration of *distinct languages* By virtue of our DSL technology, GnoSys is a multi-linguistic platform. Contracts provide key infrastructure for ensuring that components written in different languages respect the requirements of every other component's language at a language boundary. That is, contracts provide a dynamic bridge between the static requirements of different languages.

Because contracts are frequently placed on module boundaries, they ease the burden placed on our analysis and verification tools, factoring problems that require global, system-wide analysis into problems that can be solved locally.

Finally, we can use program analysis and specialized domain-specific contract languages to extract static knowledge about a program from its associated contracts. This, again, illustrates the expressiveness-first philosophy of GnoSys: contracts specified with restricted DSLs can be processed at compile time, while we preserve an "escape hatch" of general procedural encodings for properties that are too complex for static treatment.

Racket as a high-level, functional operating system The principal software artifact constructed during the GnoSys project is the Racket programming system. In some sense, the Racket language, compiler and run-time system, taken as a whole, can be considered to be a high-level operating system that provides basic OS services in the context of a functional programming language. It is a *virtual* OS, sitting on top of some low-level, high-assurance OS that provides device drivers and otherwise abstracts over some machine details. In Racket, the programming language *is* the operating system. By this, we mean that *sophisticated languages provide essentially all of the mechanisms and services that traditional operating systems provide*—but in a way that facilitates reasoning about the programs and permits more complex static mechanisms to be used.

In a sense, Unix and the C programming language are symbiotic. Unix is dependent on C, in that C is the language used to implement Unix. But C is dependent on Unix, in that it was co-designed with Unix, and requires the run-time services provided by Unix. For example, Unix makes up for C's lack of memory safety by providing separate, protected address spaces for cooperating agents (which were traditionally written in C), and provides the mechanism, pipes, by which these agents communicate and cooperate. To give another example, Unix's SIGPIPE signal papers over C's lack of exception handlers, providing for the global shutdown of a pipeline of cooperating agents when one of these agents terminates unexpectedly due to an error.

This raises the question, which we address with Racket: what is the structure of an operating system that is symbiotic with a modern programming language—one providing many elements not found in C: higher-order functions, contracts, continuations, memory safety, exceptions, modules, and so forth? To answer this question, we must restructure both the operating system and the applications built on top of it in terms of the mechanisms provided by modern functional languages. In the following subsections, we'll consider multiple views of an operating system, and show how these views are reflected in the language-oriented Racket system.

Operating systems and modularity An operating system is a kind of crude, awkward module system. It provides two modules per application: kernel and user. The data structures and code of each module are separated from the other, and cross-module communication happens in carefully controlled ways.

This two-compartment division is, of course, very coarse, and leads to the all-or-nothing vulnerabilities of the monolithic kernel: because all data that must be protected from the user-space code is lumped together in the "kernel" module, once an attacker has broken into the kernel, every piece of kernel-secured data is laid bare.

The means by which these two "modules" are integrated is similarly coarse and awkward. The data typically passed across the user/kernel boundary are simple and passive; the exceptions that are higher-order (e.g., packet-filter interfaces) are rare and difficult to use. The preconditions and invariants attached to module interfaces (that is, to system calls) are informally specified and not subject to any kind of automated static checking. Thus, instead of being able to check correctness statically, interface preconditions must be left as dynamic checks. There can be no static assurance; modularity is provided by expensive, heavyweight run-time artifacts.

It's not enough simply to split a kernel into modules. This was attempted by the microkernel work in the late 1980's. Microkernels were an attempt to modularize operating systems in a programming language, C, that had no support for abstraction and modularity. As a result, the OS was factored using a heavyweight, dynamic mechanism: separate processes. Cross-module control and data flow thus involved expensive context-switch transfers, which proved to be unacceptable in practice. Microkernels did not take off because their designers did not have the tools they needed to realize their robust, modular architectures.

In GnoSys, we abandoned the simplistic kernel/user division. Once we had broken the operating system into multiple components, we could manage them with the same expressive language technology we use to manage all GnoSys components. The contracts at the boundaries of the components provide run-time assurance that kernel invariants are maintained. Contracts provide semantic specifications for reasoning tools that provide assurance.

Likewise, the compiler manages module isolation for kernel components just as it does for all components, eliminating the run-time cost of modular structure. In particular, by handling module isolation at compile and link time, the compiler is free to optimize *across* module boundaries—even when the boundary lies at the

user/kernel interface; even when the data flowing across the interface is higher-order.

In short, GnoSys committed to the thesis that modularity and abstraction are architectural elements best left to programming languages and their associated tools, where they can be checked, optimized and otherwise processed.

Operating systems as models of computation An operating system is an abstract model of computation: we have file systems instead of hard drives; threads, instead of CPUs; and so forth. An application written for the Unix operating system is coded to an abstract specification that permits it to be executed on multiple distinct hardware platforms.

A programming language is also an abstract model of computation, but with a key teleological distinction. It is an abstraction that is adapted to *static reasoning*: programs are analyzed and otherwise processed at construction time, while operating systems are purely dynamic artifacts.

When we provide OS structures via linguistic mechanisms, we expose them to the suite of static-reasoning engines that are a part of program-language implementations. The compiler can work together with the operating system to provide system services.

In GnoSys, this specifically becomes a task of modelling operating-system structures in a functional language. For example, consider continuations, a notion from lambda calculus that is a central concern of optimizing compilers for functional languages. Continuations are an abstraction—in the operating-system sense—of CPU state; asynchronous continuations abstract the state of a preempted process. This abstraction naturally leads to a complete picture of the processor resource: hardware interrupts become abstract events, and processors become continuation transformers. It is possible, given this representation, to design virtualizable thread schedulers that operate purely with abstractions based on the lambda-calculus. These schedulers can be run in a nested fashion: a user application can write an application-specific, preemptive scheduler to manage sets of threads using CPU cycles that are passed to it by some superior scheduler. At the root of this tree of virtualized schedulers, we have the real hardware, managed with *exactly* the same kind of scheduler code.

Another example of modelling OS mechanisms in a functional language is the representation of network-protocol stacks. As we will see in a following section,

we were able to implement these, during the GnoSys project, using communications-oriented domain-specific notations that facilitated the compositional construction of high-assurance implementations.

Operating systems as resource guardians An operating system provides guarded access to the machine resources, limiting both the way that resources are used and the quantity of resources that are consumed by a part of the system. By directly encoding the OS services in the programming language, the compiler can assist in management of these resources, provide more flexible rules about their use, and provide more fine-grained accounting for the consumption of resources. For example, traditional operating systems provide "file descriptors," which are essentially references to OS-protected resources. All manipulations of these resources by the user code must occur at arms length across a context-switch barrier, which forces the operating system into a particular way of tracking process-specific file-descriptor use and a particularly narrow capability of processes to trade file descriptors. Similarly, the memory resource is provided by the operating system to the user code at page-level granularity, which does not match well with the fine-grained allocation needs of typical user programs. Worse, the access guards the operating system can place on memory are crude and simple, consisting only of read, write and execute capabilities on page-sized blocks of memory.

Racket, in contrast, can use the language's contract machinery to provide more sophisticated, application-oriented constraints on the way clients may use a resource. Contracts can encode a kernel's requirements on the use of a resource, but also allow layers to add their own contracts or allow applications to apply contracts on the use of a resource that is shared with other applications. Contracts on high-level values replace the parsing of byte streams to ensure that components communicate properly. Domain-specific languages allow programmers to specify such contracts in terms that are best suited to the task.

Advanced program analysis for higher-order programs

Although GnoSys permits programmers to work with domain-specific notations, the core language we used for our general computing platform is Typed Racket, which is a higher-order functional programming language based on the lambda calculus. The attraction of lambda calculus as a foundation for programming languages is that it sits in a "sweet spot" for computational notations: it is simultaneously a practical programming paradigm as well as a powerful theoretical model

of computation. Because GnoSys has a design commitment to formal, automated reasoning, it was important that programs have the sort of clean semantics we get from solid theoretical foundations.

We view lambda calculus as computational glue for connecting together general computations. It's important, then, to have tools for reasoning about systems composed with this kind of structure: program analyses that work with higher-order functional languages.

The core analysis we developed for GnoSys is a novel form of *higher-order flow analysis* Flow analysis is one of the most powerful tools available for optimizing, debugging and reasoning about programs. Every serious optimizing compiler uses flow analysis to discover the information that drives program optimization. Flow analyses can also be used to perform safety analyses, help debug programs, and improve the user experience in interactive program-development environments.

The flow analysis we developed is novel in three main ways, all of which are motivated by co-design considerations with other elements of GnoSys:

- **PDA-based control abstraction**
  Traditional flow analyses are graph algorithms that consider all paths through a fixed control-flow graph. Since the set of all such paths is a regular language, we are essentially abstracting the Turing-equivalent computation represented by the program with a finite-state automaton. Such approximation was reasonable in the 1960s world of Fortran programs, where the important control constructs are conditional branches and loops, which are well modeled by this paradigm.

  Modern functional languages, however, are poorly modelled by such an abstraction. Their key control construct, function call/return, produces control traces that are *not* regular languages: the calls and returns have a nested parenthesis-like structure. Far better precision can be obtained by abstracting the computation with a push-down automaton (PDA).

  This is particularly important in GnoSys, given its systemic emphasis on modular composition. Module boundaries in GnoSys programs are crossed by means of function calls and returns. Because PDA-based analyses don't muddy and mix together caller context on return flows, they hold the promise of keeping distinct the information flow from and to the different clients of a module. This is critical, for example, when doing a security analysis on information flow: if, for example, a hash-table module is invoked by two

clients of differing security ratings, erroneously including spurious cross-flows between the two client modules would produce false positives that would render the security analysis useless.

PDA-based analyses also enable precise reasoning about resources with dynamic extent, such as stack-allocated data, or nested locks.

- **Higher-order semantics and abstract state traces**
  Classical flow analysis considers only traces through the program structure. That is, they are focussed on *control structure* to the exclusion of other elements of the program state, such as environment and data structure.

  When we shift to a higher-order setting, the analysis' *control-flow graph* morphs into a more general and expressive *abstract state graph*, which adds in abstractions of the program's data and environment structure.

The role of program analysis in GnoSys Program analysis is a technology that enables other components of the GnoSys architecture. Just as lambda calculus is general-purpose glue for composing computations, PDA-based flow analysis is a general-purpose "weak method" for composing knowledge gained from multiple sources about a program.

Besides yielding knowledge about a program to the program designer, powerful analysis is the critical means of optimizing *across* module boundaries. In a modular, component-based system such as GnoSys, these module boundaries even include what other operating systems consider the kernel/user interface, and virtual-machine abstraction layers. Thus, the ability to statically melt away these boundaries after they have served their purpose at system design and assembly time is critical. It liberates programmers to construct their programs in modular ways, knowing there will be no run-time penalty for using the GnoSys structure.

Capturing design knowledge with domain-specific languages

Perhaps the single most powerful element in GnoSys for raising the level of programmer expressiveness is that programmers can easily design and use custom, domain-specific notations for their programming. Our approach to this programming paradigm grows out of the Lisp and Scheme macro experience, where it is part of everyday programming to design "little languages" that are carefully customized to the particular application domain being programmed. The technology is quite lightweight: a programmer can implement an entire new language with

a day of work and a few hundred lines of code. Once defined, programmers can switch between languages simply by entering an open parenthesis and a keyword; the matching close parenthesis switches the notation back to the surrounding language.

For example, the Scheme expression (written in the systems-programming system, scsh)

```
(lambda (printer)
  (& (| (gunzip) (html2ps) (lpr -P ,printer))
     (<< ,(rx-subst
           (| "John" "Paul" "George" "Ringo")
           (run/string (wget -O - http://reviews.com/letitbe))
           pre "Beatle" post))))
```

shifts between three distinct languages on a line-by-line basis: the general-purpose functional language Scheme, a special "Unix shell" notation for specifying pipelines of processes (lines 2 and 5), and a regular-expression notation (line 4) for specifying string matches.

Because domain-specific languages are used in specialized contexts, they frequently can be *restricted* in their computational power—which, consequently, *enhances* our ability to reason about the computations they describe. That is, these custom notations typically have rich static semantics, which can be mined out by program analyzers.

Consider, for example, two ways of specifying a string matcher: with a short, one-line regular expression, or with the equivalent finite automaton written in C, requiring perhaps a page or two of code. Of course, the regular expression is a clearer way to program: it's easier for the programmer to write; easier to read and understand; easier to recognize that the expression matches exactly what we intended it to match.

Not only is the regular expression easier for a human to understand, it is also easier for automatic tools to analyze. For example, it is a challenging task to determine if the C code terminates on finite input—our program analyzer is up against the halting problem. But this is a trivial task to carry out *once the computation has been encoded in the regular-expression form*: all finite automata halt on finite input. By using a domain-specific language, we have captured knowledge about the program *for free*—it requires no work from the programmer. By working in the DSL, the programmer works less, but says more.

All of this assumes, of course, that the extra static semantics associated with DSLs can be made available to program-analysis tools. This is not, unfortunately, true of the classic Scheme and Lisp macro technology: these linguistic tools only provide clarity of notation to human programmers. The GnoSys team extended this technology to permits us to attach this extra static semantics to the custom notations designed by programmers for their applications.

3. Methods, Assumptions and Procedures

Except in ways that will be discussed below, our methods are the standard methods of academic research: the primary unit of investigation was a professor-advisor/doctoral-student-advisee pair, carrying out some particular thrust of exploration. The project also had two post-docs, Tobin-Hochstadt and Van Horn (who are now both professors, at Indiana and Maryland, respectively, doing work for DARPA under their own contracts). Thus, much of our work was quantized into doctoral-dissertation-sized chunks. The various results were published in peer-reviewed conferences and journals, both to disseminate the work but also to subject it to the scrutiny and rigor of peer review. This is pretty classic methodological stuff, but it's robust and produces quality science.

On the other hand, our work is also realized in the construction of software artifacts, and here, our methodology is not quite so standard. Engineering research, such as Computer Science, often centers on the construction of experimental prototypes: expensive, laboriously hand-constructed one-off systems that are used to evaluate theories; once the data is gathered, the system is discarded—the final result is the *data*, not the *artifact*. Thus, the researcher has to build something that is "toy" enough to be built on time and under budget by a small team, but "real" enough for the data gathered to constitute compelling evidence for some engineering claim.

In GnoSys, we did things differently. In brief, we use what we build. While we did build some research toys as initial prototypes, we frequently scaled these first implementations into full-on, delivered software: technologies developed under the GnoSys contract were rolled out in successive releases of the Racket programming environment throughout the four years of the project.

In particular, our work on contracts, gradual typing and domain-specific language-development technology (that is, hygienic macros) are all now in daily use by programmers around the world—Racket is publically available and open source. Internally to Northeastern and the University of Utah, the programming environment is used by doctoral students and professors to do research, as well as by hundreds of college freshmen learning the introductory rudiments of programming. Outside our the GnoSys-project home institutions, it is used all over the world for teaching as well as industrial software development. The research benefits of this kind of methodology are clear: the ideas embodied by the software get much, much more thoroughly exercised and evaluated, across a wide spectrum of use cases.

4. Results and discussion

We've broken these results out, but the whole is greater than the sum of the parts. All of this work happened in a common context, with the explicit goal of weaving them together into a unified whole.

Racket on a router

As a testbed and driver application for GnoSys technology, we ported the Racket programming system to a commercial, consumer-grade wifi access point, the Netgear WNDR3700 router. We were able to compile, link and execute Racket code directly on the router. This includes all the advanced elements of the language: first-class functions, full continuations, concurrent threads, the garbage collector, the full module system, the full contract system, all of the language tower, and access to the underlying TCP/IP stack.

Following this, we implemented a suit of network services on the Racket platform: an ssh server permitting one to log into a Racket read-eval-print interaction, two forms of DNS server (a root server, and a caching proxy server), and a chat/instant-message service. The caching proxy DNS server has been in daily use in the College of Computer Science for several years now, running on a wifi access point that provides service to our floor of the building.

All of the various pieces of technology we developed for CRASH synergistically supported this application development. For example, our work on program analysis and optimization of the Racket's numeric types made the system acceptably performant. Contracts kept the system secure, at the virtual operating-system level. We used macro technology to create domain-specific languages that let us declaratively describe packet layouts; macros then automatically produced marshalling and unmarshalling code for the packets, eliminating buffer-overflow and other bug-exploiting security attacks.

One of the reasons we did the Racket-on-a-router work was to provide the security people at Lincoln Labs a means of evaluating our security and robustness improvements without requiring them to program in Racket themselves. A network protocol is a powerful abstraction barrier: by providing them with a box running these services, they could evaluate at the protocol level.

Our services closed essentially every known exploit on the DNS and SSH services. Bottom line: it's crazy to continue implementing these kinds of critical services in languages such as C.

Compositional asynchronous architecture

Based on this experience gained building network services directly in Racket, we developed a software architecture, delivered on top of Racket, specifically for developing network services. Our "Compositional Asynchronous Architecture" has a combination of features that mesh together to facilitate the construction of network servers:

- a pub/sub system built around a tree-structured "topic" space;
- a self-virtualizing abstract machine that controls and constrains communications interactions;
- a set of system-provided messages which includes both "presence" and "absence" of potential pub/sub interactors for a given topic. Providing absence notification is a key functionality that makes it much easier to build robust services backed up by redundant servers.

We carried out a study of the architecture by using it to re-implement all of our previous direct-in-Racket network services: ssh, root DNS, proxy DNS, and a chat/instant-message server. We were then able to shape this architecture into a core calculus, Minimart, and a new domain-specific programming language, Marketplace, for developing robust, high-assurance (but performant) network services. Marketplace is the topic of a forthcoming doctoral dissertation (and will be rolled out in the general release of Racket in the near future).

Alan Kay's Viewpoint Institute invited Tony Garnock-Jones, the graduate student carrying out this work, to visit for six months this past year to investigate the synergy between the GnoSys Network Calculus work and their work on "networks all the way down."

Eliminating C from Racket

At the beginning of the CRASH program, we realized that the Racket run-time system was implemented using more C than we'd like, given the CRASH emphasis on high-assurance systems. Our entire approach to high-assurance is based on exploiting high-level languages; code written in C is outside the domain of our technologies. We spent a fair amount of work hardening up Racket by rewriting entire subsystems of the Racket implementation from C to Racket.

In particular, we completely redesigned and rebuilt Racket's metasyntactic programming facility, its hygienic macro system. The original one, which has been deployed for over well over a decade, had been written in C, for performance reasons. It has seen hard, industrial-scale use and performed well, but we were uncomfortable with the fact that such a critical, core element of the entire Racket platform was constructed as a very complex piece of C code, where none of the Racket technology could be looped around and brought to bear on the implementation.

Metaprogramming and DSL technology

This is the foundational work that lets us develop language-oriented solutions to the programming tasks we tackle in GnoSys: advanced macro systems are the technology we use to construct domain-specific languages (such as the Marketplace communications-oriented programming language, above), and to implement languages with extended static semantics (such as contract systems, used to capture programmer knowledge at software-construction time). Domain-specific languages are a key means of raising the level of discourse: programmers can design custom notations purpose-built for a given application, and the information is expressed in a means that is accessible to the language-processing technology, rather than hidden in comments or encoded in obfuscatory idioms or patterns of code.

Our DSL technology is, again, something that we use everyday in our program-development work. At one count during the CRASH program, we were using over 40 domain-specific languages in the implementation of Racket, and the technology is widely used by Racket users outside the GnoSys group.

Scopes as sets of binders

As mentioned above, one of the straightforward engineering tasks we carried out over the course of the CRASH effort was the reimplementation of Racket's macro system in Racket itself, instead of C. This permitted us to move ahead with a new architecture for maintaining "hygiene," that is, the compile-time structures available to the DSL implementor for managing variable scope and variable capture.

While we were doing this reimplementation, we were able to completely revisit the core hygiene algorithm. Hygienic Scheme macro processors are an arcane and ill-understood branch of metaprogramming technology; when these systems are extended to work with module systems and sophisticated phase distinctions

(as Racket's is), they become even more complex. Redesigning this system advanced our understanding of the interactions between hygienic macro expanders, module systems and phase distinctions.

The new design, which took several years to design, prototype and release, is based on a notion of scope as consisting of sets of binders or "binding contours." The new system is much simpler and more straightforward. A paper describing the new mechanism was presented at POPL in January of 2016, just as CRASH was winding down. The new system is now deployed in the current release of Racket; users have reported that they are quite happy with it.

Hygiene and nominal types

We have also performed foundational work on adapting nominal types and nominal logic to develop an implementable theory of hygiene for next-generation macro systems.

A key problem in meta-programming and the development of domain-specific languages is *binding-safe* term manipulation—that is, manipulating terms (or syntax trees) in such a way that variables are guaranteed to refer to their correct binders, neither escaping from their scopes nor being introduced into scopes where they don't belong. Existing methods for dealing with this problem either impose unrealistic constraints on syntax or depend on dynamic solutions that do not give definition-time guarantees.

In the early days of GnoSys, one of our graduate students, David Herman, for his doctoral thesis, developed a system for the specification of binding patterns, and gave an algorithm for determining (at macro-definition time, not at macro-use time) whether a macro defined using patterns and templates would respect those binding patterns. The technical development was based on the idea of α-conversion; that is, systematic renaming of bound variables and their references. Each binding specification gives rise to a notion of α-conversion on its terms, and binding-safe programs are ones that take α-equivalent inputs to α-equivalent outputs.

Herman's dissertation advanced the state of understanding of these mechanisms. In fact, it was the first coherent theory explaining what hygiene was and meant, twenty-five years after the introduction of the idea. But it was not practical to deploy for real programming: the price of the static guarantees was unrealistic restrictions on the expressiveness of the system.

After Herman's dissertation was completed, we turned our attention to extending his ideas to macros defined by tree-manipulating procedures. Our new design was based on Pottier's Pure Fresh ML, but with a far richer language of binding patterns, and with stronger guarantees than those of Pottier. A program written in the language is analyzed to generate proof obligations, which are discharged automatically by an SMT prover (in our case, Z3). Because new variable names must be generated, programs in the language are non-deterministic.

The next round of work we did in this area under CRASH was establishing the theoretical underpinnings of the new, more powerful and expressive mechanism. In particular we needed to establish

1. Soundness: Given two α-equivalent inputs, a program that passes its static checks can only produce α-equivalent outputs.

2. Determinacy: For any input, all the possible outputs of a program that passes its static checks will be α-equivalent (and therefore, all strategies for generating fresh names yield equivalent answers)

Because of the richness of the binding-specification language, each of these presented a formidable technical challenge.

By the end of 2013, we had completed the proofs of these theorems, and began turning this core mechanism into a proof-of-concept prototype language, named Romeo. We published a paper on this work in ICFP.

In a follow-up redesign of Romeo, we were able to produce a more user-friendly language, Romeo-L, with a better surface syntax, and with a better connection to the SMT backend. An outstanding feature of Romeo-L is error localization, which translates failures from the SMT back into messages that convey this information in terms of the user's program.

The Romeo-L work was a Master's thesis; the larger Romeo effort has been a doctoral thesis whose defense is scheduled for April of this year. Eventually, we hope to have this technology rolled out into general use in Racket.

We should add that Herman, who started this original line of work under CRASH funding for his doctorate, is now Chief Technology Officer at Mozilla, where the Rust programming language is being developed. Rust is being designed to have hygienic macros, so there is a clear path to exporting the technology developed under CRASH out beyond the Racket community.

Contracts and Types

GnoSys exploits "contracts" to attach what amount to security and safety checks to software modules. We have multiple technology results related to contracts:

- The design of analyses to check and discharge contract checks at compile time: safety without cost.
- New run-time support ("chaperones" and "impersonators") to make the remaining overhead for contract checking lightweight.
- New developments in "blame theory," the mathematics associated with contracts that provide the ability to identify the responsible agent when a complex contract is violated.

We have also done significant work extending the typing mechanisms of Racket to work on the full language. This includes Racket's class-oriented object system, as well as its powerful, difficult-to-type "delimited control" operators. The delimited control operators are particularly relevant to the goals of CRASH, as they give programmers the ability to tightly constrain non-local control transfers in subsystems. (For example, when beginning students enter Racket code into the Dr.Racket development environment, Dr.Racket's evaluator uses delimited-control operators to prevent runaway, buggy code from crashing or taking over the system.)

This is an example of where we were able to achieve the kind of cross-layer optimization and restructuring of the system stack that was part of DARPA's charter for CRASH. The Racket run-time system is a kind of virtual operating system, responsible for allocating and controlling resources to various systems implemented in Racket. For example, programs require both processor cycles and memory to execute; Racket not only hands out these resources to client systems, but controls access to them. Delimited control operators are the language-level means by which we enforce control constraints on the various systems that run on top of Racket.

By providing these mechanisms at the language level, we were also able to subject them to program analysis. Again, here we achieved cross-layer synergy: as we'll describe below, our work on PDA-based flow analysis was able to handle analysis of programs that used these mechanisms.

We also did work extending the expressiveness of the contracts that Racket provides. *Behavioral* contracts supplement interface information with logical assertions, often written in the same programming language as the component itself. *Temporal* contracts further enrich interfaces with a language for specifying adherence to stateful protocols. Together, these assertions can provide strong invariants that are monitored at run-time, giving a precise explanation of blame when failures occur.

Contracts offer significant software engineering benefits, but they come with certain costs. Contract monitoring can be expensive, particularly in the case of higher-order values, which must be wrapped as they flow across component boundaries, and temporal monitoring significantly increases the run-time burden. We invested a fair amount of work into the design and implementation of technology to make the run-time overhead of contract enforcement lightweight. This new technology is now provided in the standard release of Racket.

Graduate student Dimoulas, with others in the GnoSys team, introduced the notion of "option contracts" because full-fledged contracts remain expensive. In analogy to the business world, option contracts are the right to impose a contract, but they are not a contract. With option contracts, programmers can dynamically add/remove contracts as trust for a component grows/shrinks. This gives us a kind of "security knob" that we can turn up or down as circumstances warrant, trading off performance for safety.

PDA-based analysis for higher-order languages

GnoSys has been developing powerful program analyses based on push-down automata models of computation. These models give strikingly more precise results for languages where function calls are the dominant control structure (such as the functional programming languages we use).

The languages we use in GnoSys have additional elements, called "first-class control operators" that permit extremely general control transfers in the program: `call-with-current-continuation`, `prompt` and `reset` are three examples of such operators. They are particularly important in the systems-programming context of GnoSys, as their use tends to pop up in the context of "operating-system-like" mechanisms, such as thread schedulers, and language elements for "locking" software systems into control-bounded sandboxes. We rely upon these operators in the virtual operating system we've provide in Racket, to enforce strong,

language-based resource barriers for systems. This is part of the cross-layer re-architecting of the computational stack we've explored for CRASH: we use language elements (like `prompt` and `reset`) to provide the kind of application security and protection that other operating systems provide with separate address spaces and, say, the `fork()` system call.

General framework for higher-order abstract interpretation

We've also published work on "monadic abstract interpreters." This is a result that will make it much easier for others to build custom analyses using our techniques. The problem addressed here is that our PDA-based techniques are pretty esoteric. The "monadic abstract interpreter" result shows how one can package up these analyses as a set of building blocks that can be assembled into a complete analysis. It represents a kind of "unified theory" that encompasses a wide range of analytic techniques: polyvariance, context-sensitivity, flow-sensitivity, heap-cloning, etc. Our entire analysis framework is parameterized so that monads may be swapped in and out to determine flow-, path- and context-sensitivity—all for the kind of advanced and difficult-to-analyze programming languages we use in the GnoSys project.

We then took our initial work on mathematical frameworks for doing advanced analyses of higher-order languages and captured this framework in a language itself: a domain-specific language making it simple to specify sophisticated, task-specific analyses for higher-order languages such as Racket.

Doing so provided two big benefits. First, it improved correctness: it is much easier for non-specialists to develop analyses of interest and get the implementation right. The kind of analyses we've been developing over the lifetime of the CRASH program are subtle, arcane and easy to get wrong. Packaging up these algorithms in a DSL means that a specialist can get the general algorithm right, once, and non-specialist consumers can plug their own analysis problems into these algorithms without having to understand all their intricacies.

The second benefit is efficiency. By specifying our analyses in a DSL, we were able to subject the specification to automatic optimization, producing analyses that run, not 10% or 15% faster, but 2-3 orders of magnitude faster. In the first several years of CRASH, we worked out techniques for doing so, by hand. In the final phase of CRASH, we did the DSL work that automated it. Providing these speedups is important: PDA-based analyses are very precise, but can be unworkably expensive to apply to large code bases. What we got was a system

that had no tradeoffs: it provided analyses that were faster *and* more precise *and* easier to implement.

Might, Van Horn and graduate student Johnson worked out techniques for analyzing higher-order control operators using our PDA analysis. These operators are very, very difficult to analyze, as their whole *raison d'être* is to provide non-standard paths of control, which deeply affect the control- and data-flow of the computation. We had worked out ways to handle these control operators in our analyses in the early days of GnoSys, but these solutions were *ad hoc*, and didn't fit into our general "Abstracting abstract machines" analysis framework—hence, they did not fit into the tools we were constructing based on this framework. By the end of the CRASH program, we were able to do this kind of analysis within the general framework. This final result was the culmination of work carried out throughout the entire project.

Package management

We designed, implemented and released a new package manager for Racket; it is now in use in the general Racket system.

Racket's package manager reflects the "raising the discourse of programming languages" theme of GnoSys: packages include documentation that can be intertwined with the source code and require compilation and possibly execution steps to produce. Not only is the documentation embedded within the source code, but *vice versa*, which is a way to ensure that the documentation does not become stale or invalidated by code changes. Note that Racket documentation is written in one of Racket's many domain-specific languages; as this language can switch into general Racket, the documentation language is Turing-complete.

We are working on technology to build binaries and documentation for potentially untrusted packages. In keeping with the "eat your own dogfood" practices we've observed throughout GnoSys, this work is being channeled into Racket's general package manager—this is how we stress our designs to find issues with them.

The new package-build service in Racket's package manager handles two key issues: (1) Racket's powerful and dynamic build-phase facilities raise security concerns not just when *executing* foreign code, but also simply when *building* foreign code; and (2) the documentation for Racket packages is deeply intertwined with its source code, being written in one of Racket's domain-specific languages (one which permits general-purpose computation).

Based on our experience constructing this package system, we are developing a general model of build systems and package managers, to make the lessons learned available for other programming languages.

Profiling secure code

Dynamic languages (such as Racket) are well suited to profile-driven optimization. We made significant progress on improving and deploying Racket's feature-specific profilers. While the original plan called for the use of such a profiler to determine the cost of interactions between typed and untyped components in a system, we found a second exciting use for the technology during the CRASH program. Graduate students St-Amour and Andersen applied the profiler to Dimoulas and Chong's SHILL secure scripting language (at Harvard University), which is implemented in Racket. They were able to diagnose bottlenecks in the interaction between OS-level sandboxes and SHILL programs.

This is another example of the kind of OS/application cross-layer win in the spirit of the CRASH charter. Andersen, St-Amour, and Felleisen are preparing a paper on this work and its case study.

Optimization coaching

Because GnoSys is centered on the use of expressive languages, an important component of the project was performance optimizations to make it possible to use these languages in resource-sensitive application domains. One of the subprojects of GnoSys that addressed this need was the "optimization coach," a compiler phase that engaged the programmer in a dialogue aimed at optimizing his program. The optimization coach had "meta" knowledge of the optimization task, and was able to compile lists of possible optimizations that were blocked due to insufficient knowledge on the part of the compiler. The coach would make suggestions to the programmer for ways to alter the code to enable the blocked optimizations.

This technology was used by Racket programmers outside the GnoSys project to optimize application codes. In some case, scientific programs got integer multipliers in performance.

The optimization coach is an example of very fast technology transfer out into industry from the GnoSys project. The work was done as the doctoral thesis of a

GnoSys graduate student. Mozilla invited the student to apply his technology to their JavaScript compilers, which he did.

5. Conclusions

The GnoSys project has been an effort to weave together multiple technical threads of work that we believed would constitute the kind of cross-layer whole sought by the CRASH program. We had some successes, and we had some failures—which includes successful research results that we were unable to integrate into the whole by the end of the program.

Some of the major components that we *were* able to integrate and field in Racket are the following.

- The utility of higher-order, functional languages as an expressive substrate specifically for developing secure, robust systems software seems well established. (We're not claiming to have invented higher-order, functional programming; we simply saying that we have pushed its implementation technology forward as a necessary component of our total platform.)

- Software contracts are clearly a valuable means of capturing design knowledge—"checkable documentation"—of a kind that exceeds the boundaries of static type systems. Over the course of the CRASH program, we extended the ex- pressive power of these contracts and also greatly reduced their overhead. There's no tradeoff here: we got improvements in both areas, and the im- provements are available to anyone who downloads and uses Racket.

- We've also clearly demonstrated the value of domain-specific languages for constructing components of large, complex software systems. Racket itself is written using about 40 DSLs that all interoperate (using a module system that is, itself, another DSL) to provide a single system; the Racket documentation is written in yet another DSL implemented with the same meta-programming technology. Programming with DSLs confers not only clarity and concision, but is also a great aid to preventing security attacks, as discussed earlier in this report.

We delivered on the spirit and vision of CRASH here: these three technologies work together harmoniously to deliver secure, robust software, and they do so in the language-oriented way that we originally planned: all three help to "raise the level of discourse" in programming. That was the vision, and that's what we did.

To give one supporting example, the network router and network services that we implemented in Racket completely shut down the vast majority of known attacks on the device—and this software exploited all of the individual technologies listed above.

However, there are important elements of our original program of research where we got results that we were *not* able to weave into the whole over the lifetime of CRASH:

- Macro hygiene, a topic of study in the programming-languages for the past thirty years, is only now just beginning to have a clear underlying formal theory—courtesy of CRASH-funded work. We had two different, significant results here: the work on scope-as-contour-sets, and the work on applying nominal type theory to the problem of hygiene.

  The former work did eventually get integrated into Racket, but only in the year after CRASH had ended. It is now in daily use by the entire Racket user community—both by those who develop DSLs and other syntactic extensions, and those who program in the resulting languages completely unaware of the technology. The first paper on this technology appeared at POPL January 2016.

  The latter, more ambitious, work is even more early-stage: it has been captured in the design of the experimental prototype language, Romeo, and described in a doctoral dissertation only now out being distributed in draft form to the members of the graduate student's thesis committee. So, at the time of this final report, the meta-linguistic mechanisms articulated in Romeo have not made it out into use in any real way.

- PDA-based flow analyses remain a standalone result; We've shown that flow analyses for higher-order languages can be computed with much higher precision than heretofore possible, and that this precision can be obtained with surprisingly good efficiency. That's a good thing to have done, as an isolated, standalone research result.

  But we were unable, in the four-year time frame of CRASH, to get these powerful new analyses rolled into the Racket development environment for general use. Just bringing down the cost of these analyses was the work of an entire doctoral effort; the shift from research experimentation into daily engineering practice remains before us.

At the CRASH program wound down, we were just beginning work on using these analyses to reduce the run-time overhead of complex temporal software contracts, which was the integration point we had originally conceived five years prior when we assembled the entire GnoSys vision. This work has not stopped: it's the subject of a current graduate student's doctoral thesis and is ongoing. Static analysis and contracts were made for each other; we'll get there.

- The boundary between gradual types and contracts turns out be shockingly expensive in terms of computational overhead. Both technologies have been successful as independent mechanisms, but when programmers use both statically checked types and dynamically checked contracts, the checking overhead that must be inserted into the program when values flow across the boundaries between regions of code managed by these two mechanisms can require more processing than the actual main computation itself—sometimes much more.

  When we rolled these mechanisms out into Racket, our more aggressive external users attempted use them together, with unhappy, performance-destroying results. This was a surprising discovery; it got our attention.

  In short, these two ideas ought to go well together, but this turns out to be much, much more difficult to realize than would initially seem to be the case. As a result of the GnoSys investigations into the topic, this topic has become a research area that is receiving intense study in the research community (beyond the GnoSys team, that is). In January 2016, an entire session of the POPL conference was devoted to papers on this subject. The best we can say, as of the date of this report, is the following:

  - The simple idea that this is a difficult technical problem is a result in and of itself.
  - The problem is now receiving a large amount of attention from various researchers, which, we hope, bodes well for the future.

- Gradual types fit in with a long-term research program we have on the subject of "script evolution," that is, taking programs written in dynamic languages (such as Perl, Python, and Scheme) and gradually evolving them, on a module-by-module basis, to "harden" them up in terms of their static semantics. Programs grow organically, and it is frequently not possible to throw out a program that has grown from an initial, small script into a large,

unmaintainable mess, and start fresh. Instead, the program must be evolved, by adding statically checked annotations to the existing code in an incremental way.

It was our hope, when CRASH began, to be able to extend the gradual typing machinery we were developing over the course of the project to handle Racket's complex object-oriented language elements. This is a necessary step to transitioning the technology from Racket (and other functional languages) out to industry-standard OO languages. We are not there yet: as of the end of the CRASH program, this is an ongoing doctoral thesis—it is, again, a problem requiring more than four years of work.

7. References

This list of 55 conference and journal publications that resulted from the GnoSys project has been broken up by topic.

Macros and domain-specific languages

Parsing reflective grammars.
Paul Stansifer and Mitchell Wand.
Workshop on Language Descriptions, Tools, and Applications (LDTA), March 2011.

Languages as libraries.
Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen.
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2011.

Parsing with derivatives.
Matthew Might, David Darais and Daniel Spiewak.
International Conference on Functional Programming 2011 (ICFP 2011), September, 2011.

Romeo: A system for more flexible binding-safe programming.
Paul Stansifer and Mitchell Wand.
Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014), September 2014.

Binding as sets of scopes.
Matthew Flatt.
Principles of Programming Languages, POPL 2016.

Software contracts

Correct blame for contracts: No more scapegoating.
Christos Dimoulas, Robby Findler, Cormac Flanagan, Matthias Felleisen.
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2011.

Complete monitors for behavioral contracts.
Christos Dimoulas, Sam Tobin-Hochstadt and Matthias Felleisen.
European Symposium on Programming (ESOP), March 2012.

Chaperones and impersonators: Runtime support for reasonable interposition.
T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler and Matthew Flatt.
Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October 2012.

Higher-order symbolic execution via contracts.
Sam Tobin-Hochstadt and David Van Horn.
Object Oriented Programming, Systems, Languages and Applications (OOPSLA), October 2012.

Constraining delimited control with contracts.
Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt.
European Symposium on Programming (ESOP), March 2013.

Contracts for First-Class Classes.
T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa and Matthias Felleisen.
TOPLAS: ACM Transactions on Programming Languages and Systems 35(3): 11 (2013).

Option contracts.
Christos Dimoulas, Robert Bruce Findler and Matthias Felleisen.
Object-Oriented Programming Systems Languages and Applications (OOPSLA), October 2013.

Soft contract verification.
David Van Horn, Phuc C. Nguyen and Sam Tobin-Hochstadt.
International Conference on Functional Programming (ICFP), 2014.

On Contract Satisfaction in a Higher-Order World.
Christos Dimoulas and Matthias Felleisen.
TOPLAS: ACM Transactions on Programming Languages and Systems 33(5) (2011).

Static analysis

Abstracting Abstract Machines.
David Van Horn and Matthew Might.
Communications of the ACM, Research Highlights, 54(9), September 2011.

A family of abstract interpretations for static analysis of concurrent higher-order programs.
David Van Horn and Matthew Might.
ACM SIGPLAN Static Analysis Symposium (SAS), September 2011.

Pushdown flow analysis of first-class control.
Dimitrios Vardoulakis and Olin Shivers.
ACM SIGPLAN International Conference on Functional Programming (ICFP), September 2011.

Flow-sensitive type recovery in linear-log time.
Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan and R. Kent Dybvig.
Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October, 2011.

A case for Galois connections.
Jan Midtgaard, Michael D. Adams, Matthew Might.
Static Analysis Symposium 2012 (SAS 2012), September, 2012.

Introspective pushdown analysis of higher-order programs.
Christopher Earl, Ilya Sergey, Matthew Might, David Van Horn.
International Conference on Functional Programming 2012 (ICFP 2012), September, 2012.

Monadic abstract interpreters.
Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clark, Frank Piessens.
Programming Language Design and Implementation (PLDI), June, 2013.

AnaDroid: Malware analysis of Android with user-supplied predicates.
Shuying Liang, Matthew Might and David Van Horn.
Workshop on Tools for Automatic Program Analysis, June 2013.

Concrete semantics for pushdown analysis: The essence of summarization.
J. Ian Johnson and David Van Horn.
Workshop on Higher-Order Program Analysis, June 2013.

Introspective pushdown analysis.
Christopher Earl, Ilya Sergey, Matthew Might and David Van Horn.
Journal of Functional Programming.

Optimizing abstract abstract machines.
J. Ian Johnson, Nicholas Labich, Matthew Might and David Van Horn.
International Conference on Functional Programming (ICFP), September 2013.

Monadic abstract interpreters.
Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clark and Frank Piessens.
Programming Language Design and Implementation (PLDI), June 2013.

Pushdown flow analysis with abstract garbage collection.
J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might and David Van Horn.
Journal of Functional Programming, 24(2-3), May 2014.

Fast flow analysis with Godel hashes.
Shuying Liang, Weibin Sun and Matthew Might.
IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), September 2014.

Abstracting abstract control.
David Van Horn and J. Ian Johnson.
Dynamic Languages Symposium, October 2014.

Pushdown control-flow analysis for free.
Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might and David Van Horn.
Principles of Programming Languages (POPL), 2016.

Operating-system structures in functional-language settings

Modular rollback through control logging: a pair of twin functional pearls.
Olin Shivers and Aaron Turon.
International Conference on Functional Programming (ICFP), September 2011.

A family of abstract interpretations for static analysis of concurrent higher-order programs.
Matthew Might and David Van Horn.
Static Analysis Symposium 2011 (SAS) September, 2011.

Automated specification analysis using an interactive theorem prover.
Harsh Chamarthi and Panagiotis Manolios.
Formal Methods in Computer-Aided Design (FMCAD), 2011.

Types and gradual typing

Typing the numeric tower.
Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt and Matthias Felleisen.
Practical Aspects of Declarative Languages (PADL), January 2012.

Gradual typing for first-class classes.
Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt and Matthias Felleisen.
Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October 2012. Best Student Paper.

Is sound gradual typing dead?
Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek and Matthias Felleisen.
Principles of Programming Languages (POPL), January 2016.

Toward practical gradual typing.
Takikawa, Feltey, Dean, Flatt, Findler, Tobin-Hochstadt, and Felleisen.
European Conference on Object-Oriented Programming (ECOOP), 2015.

Networking and communications-oriented languages

The network as a language construct.
Tony Garnock-Jones, Sam Tobin-Hochstadt and Matthias Felleisen.
European Symposium on Programming (ESOP), 2014

Compositional asynchronous architectures.
Tony Garnock-Jones, Sam Tobin-Hochstadt and Matthias Felleisen.
(Currently under submission)

Formal methods

The ACL2 Sedan theorem-proving system.
Harsh Chamurthi, Peter Dillinger, Panagiotis Manolios, Daron Vroon.
International Conference on Tools and Algorithms for the Construction and Analysis of Systems, March 2011.

Integrating testing and interactive theorem proving.
Harsh Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios.
The ACL2 Workshop, November 2011.

Run your research: On the effectiveness of lightweight mechanization.
Casey Klein, John Clements, et al.
Symposium on Principles of Programming Languages (POPL), January 2012.

Software for quantifier elimination in propositional logic.
Eugene Goldberg and Panagiotis Manolios.
International Congress on Mathematical Software (ICMS), 2014.

Data definitions in the ACL2 Sedan.
Harsh Chamarthi, Peter C. Dillinger and Panagiotis Manolios.
ACL2 Workshop, 2014.

Partial quantifier elimination.
Eugene Goldberg and Panagiotis Manolios.
Haifa Verification Conference, 2014.

ILP modulo data.
Panagiotis Manolios, Vasilis Papavasileiou and Mirek Riedewald.
Formal Methods in Computer-Aided Design (FMCAD 2014), October 2014.

Quantifier Elimination by Dependency Sequents.
Eugene Goldberg and Panagiotis Manolios.
Formal Methods in System Design, August 2014.

Software for quantifier elimination in propositional logic.
Eugene Goldberg and Panagiotis Manolios.
Proceedings of the Fourth International Congress on Mathematical Software (ICMS), 2014.

Data definitions in the ACL2 Sedan.
Harsh Chamarthi, Peter C. Dillinger and Panagiotis Manolios.
ACL2 Workshop, 2014.

Other

Applying random testing to a base type environment.
Vincent St-Amour and Neil Toronto.
International Conference on Functional Programming (ICFP), 2013.

An array-oriented language with static rank polymorphism.
Justin Slepak, Olin Shivers and Panagiotis Manolios.
European Symposium on Programming (ESOP), 2014. (Best paper award)

Profiling for laziness.
Stephen Chang and Matthias Felleisen.
Symposium on Principles of Programming Languages (POPL), 2014.

Optimization coaching.
St-Amour, Tobin-Hochstadt, Felleisen
Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2012

The call-by-need lambda calculus, revisited.
Stephen Chang and Matthias Felleisen.
European Symposium on Programming (ESOP), 2012.

List of Acronyms

CRASH	Clean-slate design of Resilient, Adaptive, Secure Hosts
DNS	Domain Name Server
DSL	Domain Specific Lanuage
OS	Operating System
PDA	Push-Down Automaton
POPL	Principles of Programming Languages
scsh	Scheme Shell
SQL	Structured Query Language
SSH	Secure Shell